Module 8 - Virtual memory

Chapter 9 (Silberchatz)

Module 8: Virtual Memory

Reading: Chapter 9

Objectives:

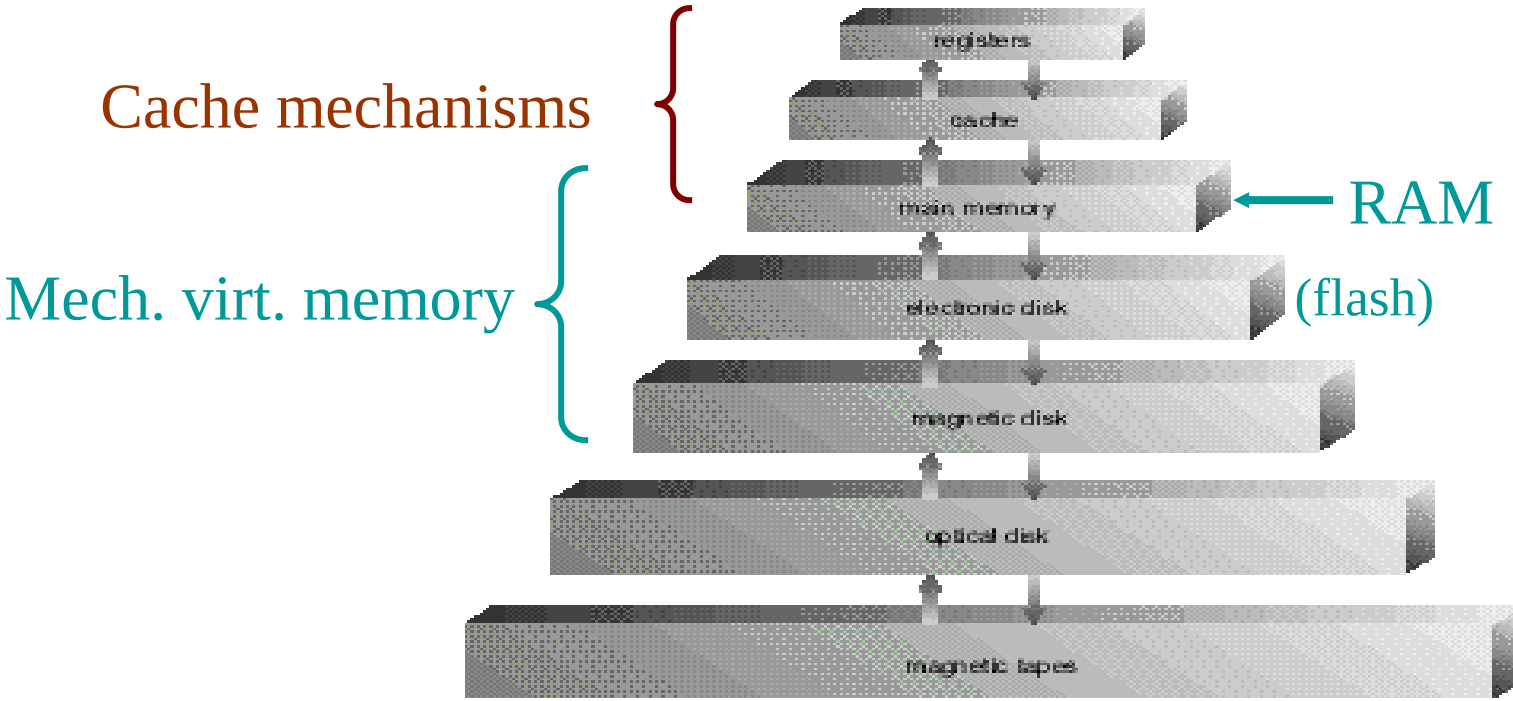
- **To describe the benefits of a virtual memory system.**
- **To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames.**
- **To discuss the principles of working-set model.**

Virtual memory

- Demand paging
- Performance issues
- Replacing pages: algorithms
- Allocation of memory frames
- Working set

Virtual Memory and Memory Hierarchy

- **Mechanisms used with virtual memory are similar to those used for cache memory**
  - But for cache memory – implemented in hardware



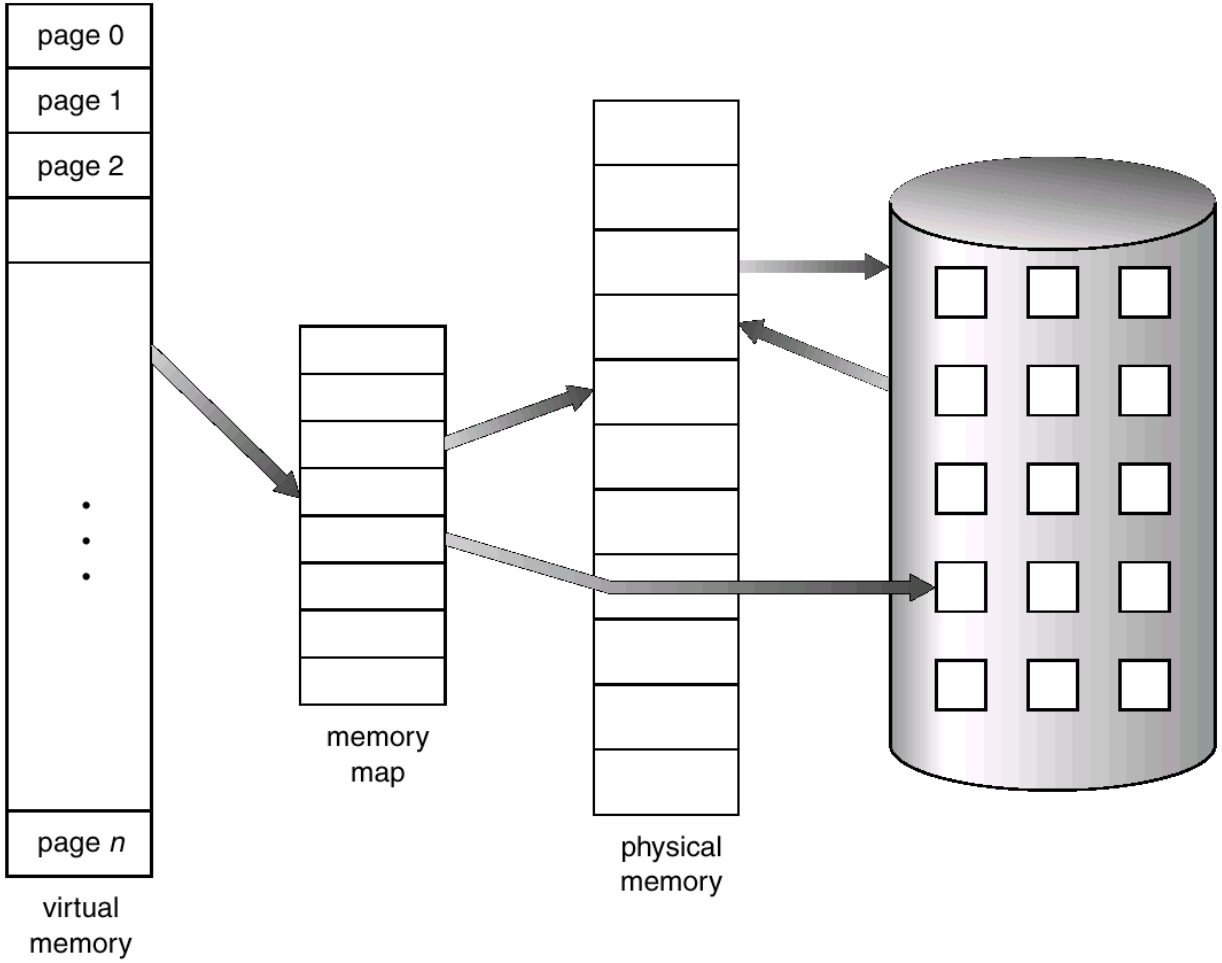
Virtual memory

- **In order for a program to be executed, it does not have to be all in main memory!**
- **Only the parts that are in execution need to be in main memory**
- **The other parts can be on secondary memory (e.g. disc), ready to be brought into main memory on request**
  - swapping mechanism
- **This makes it possible to run programs much larger than physical memory.**
  - Achieving a virtual memory that is larger than physical memory.

From paging and segmentation to virtual memory

- **A process consists of pieces (pages or segments) not requiring to occupy a contiguous region of main memory**
- **References to memory are translated to physical addresses during execution**
  - A process can be moved to different regions of memory, also secondary memory!
- **Therefore: all the pieces of a process do not need to be in main memory during execution**
  - Execution can continue as long as the next instruction (or data) is in a piece found in main memory
- **The sum of the logical memory of the running processes can therefore exceed the available physical memory.**
  - This is the basic concept of virtual memory
- **An image of the entire process address space is kept in secondary memory (normal. Disk) from where missing pages can be taken if necessary**
  - swapping mechanism

Virtual memory:
result of a mechanism which combines main memory and secondary memory



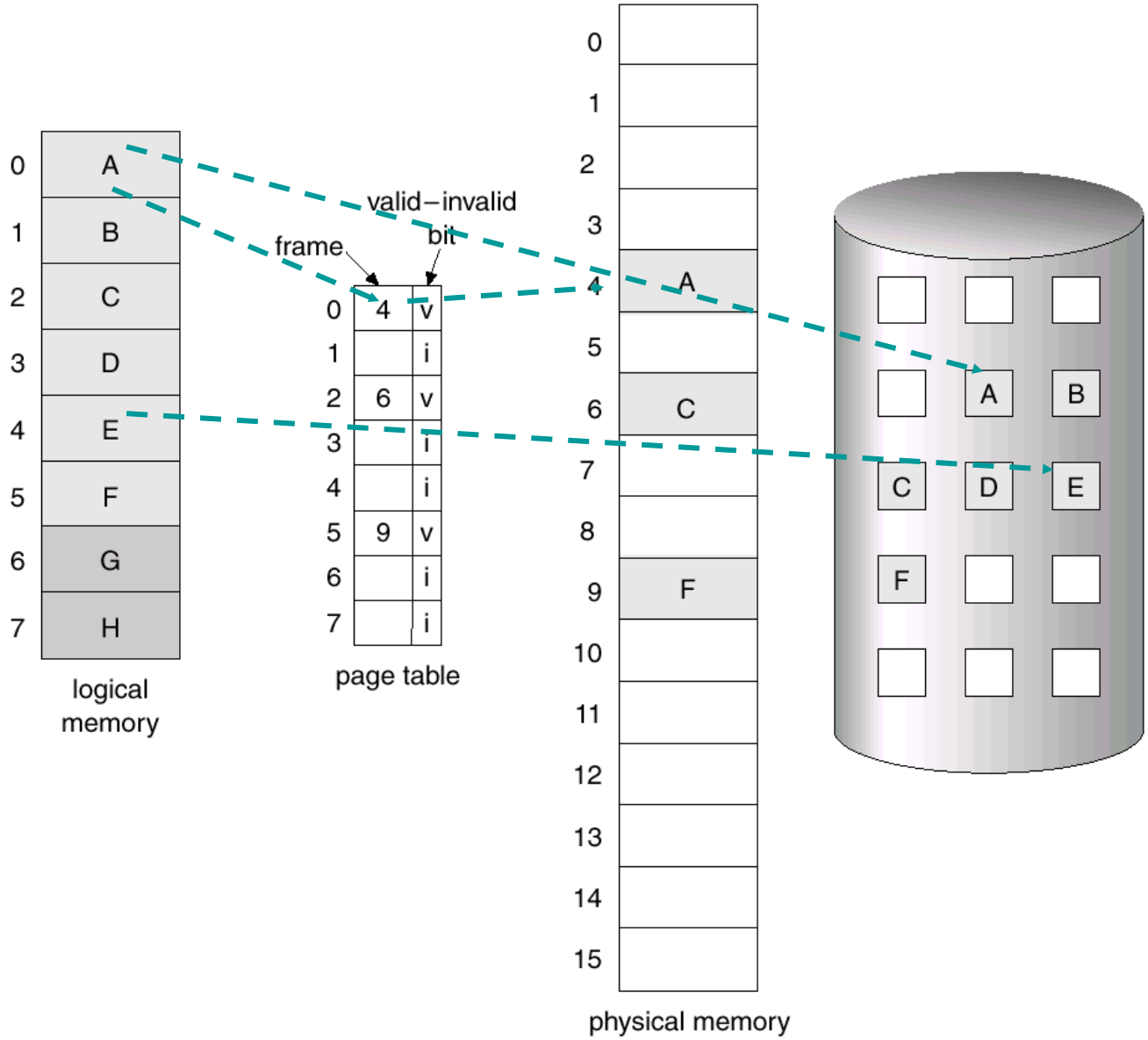
Locality and virtual memory

- **Principle of locality of references:** references to memory by a process tends to be grouped.
- Thus: only a few pieces of the process are used during a small period of time (pieces: pages or segments).
- **There is a good chance of "guessing" which pieces will be in demand in the near future.**

Pages in RAM or on disk

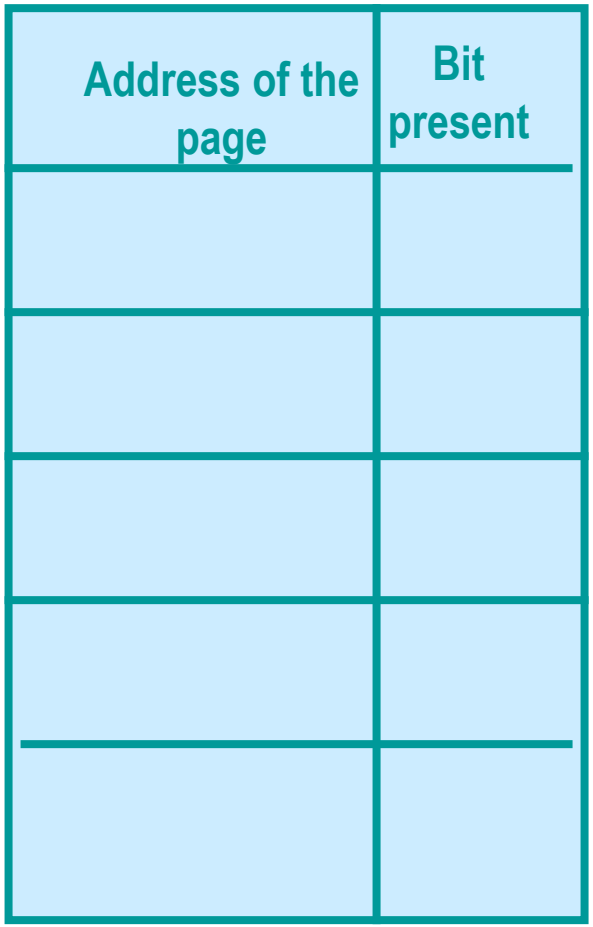
Page A in RAM **and** on disk

Page E **only** on disk



New page table format (the same idea can be applied to segment tables)

If the page is in main mem., this is an addr. of page in Main mem. otherwise it is a secondary memory address



bit *present*
1 if in main mem.,
0 if in second memory.

At the start, bit present = 0 for all pages

Advantages of Partial Loading

- **More processes can be maintained in memory for execution**
  - **Only a few pieces are loaded for each process.**
  - **The user can be happy, since he/she can execute many processes and reference large data structures without worrying about filling main memory.**
  - **With many processes in memory, more probable to have a process in the ready state, which increases utilization of the CPU.**
- **Many pages or segments rarely used need not be loaded at all into memory.**
- **It is now possible to execute a set of processes even if their sizes exceed the main memory**
  - **It is possible to used more bits in the logical address than the number of bits used for addressing main memory.**
  - **Logical address space > > physical address space.**

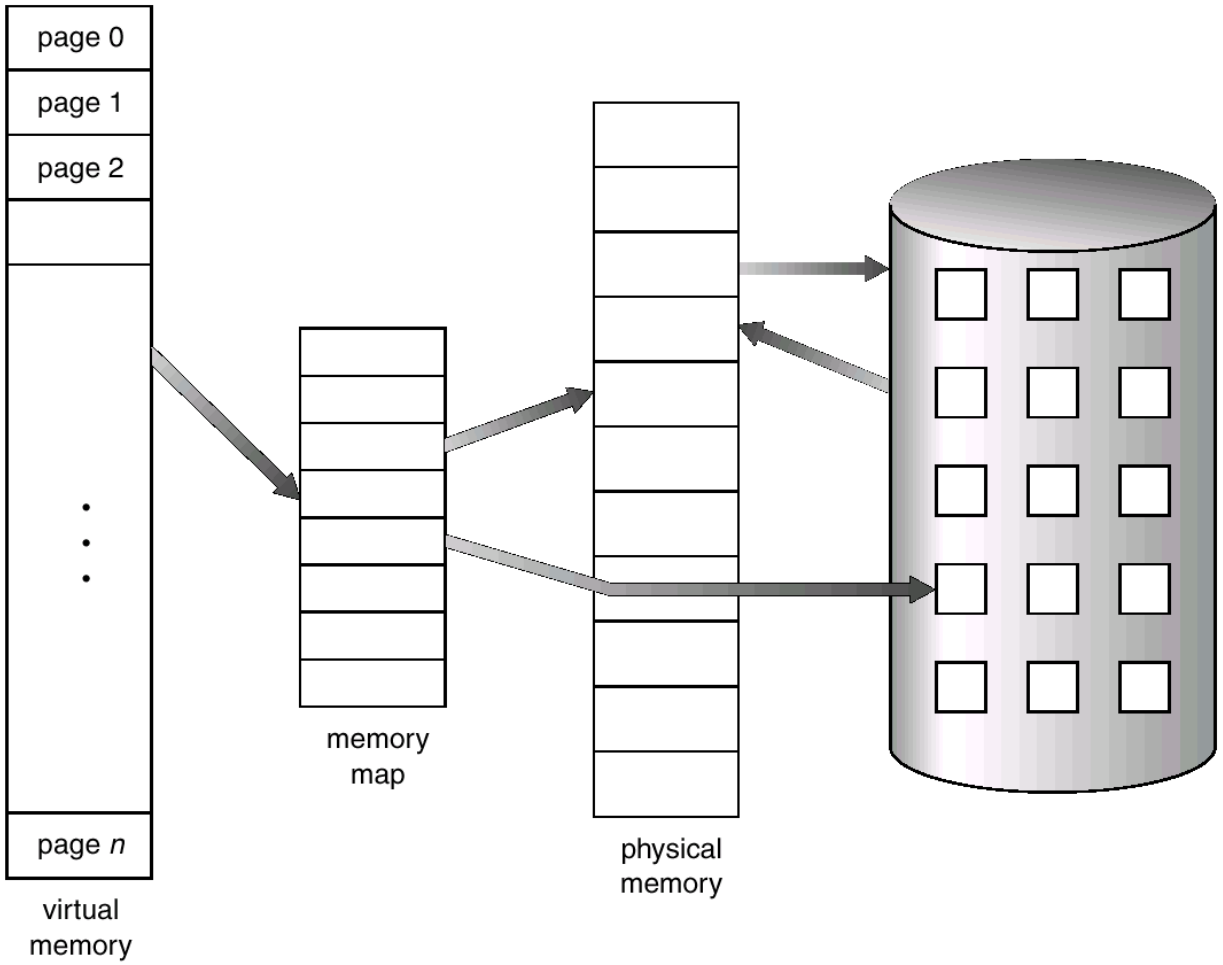
Virtual Memory: Can Be LARGE!

- **Ex: 16 bits are required to address physical memory of size 64KB**
- **Using 1KB pages, 10 bits are required for offset.**
- **For the *page number* of the logical address, it is possible to use more than 6 bits, since not all pages need be in memory.**
- **Thus the limit of virtual memory is defined by the number of bits that can be reserved for the address**
  - **In some architectures, these bits can be included in registers.**
- **Logical memory is thus called *virtual memory.***
  - **Is maintained in secondary memory**
  - **Pieces are brought into main memory only when necessary, on demand.**

Virtual Memory

- **For better performance, virtual memory is located in a region of the disk not managed by the file system.**
  - **Swap memory.**
- **Physical memory is the memory referenced by physical addresses**
  - **Found in RAM and the cache.**
- **The translation of the logical address to the physical address is accomplished using the mechanisms studied in the previous module.**

Virtual memory: the reciprocating mechanism



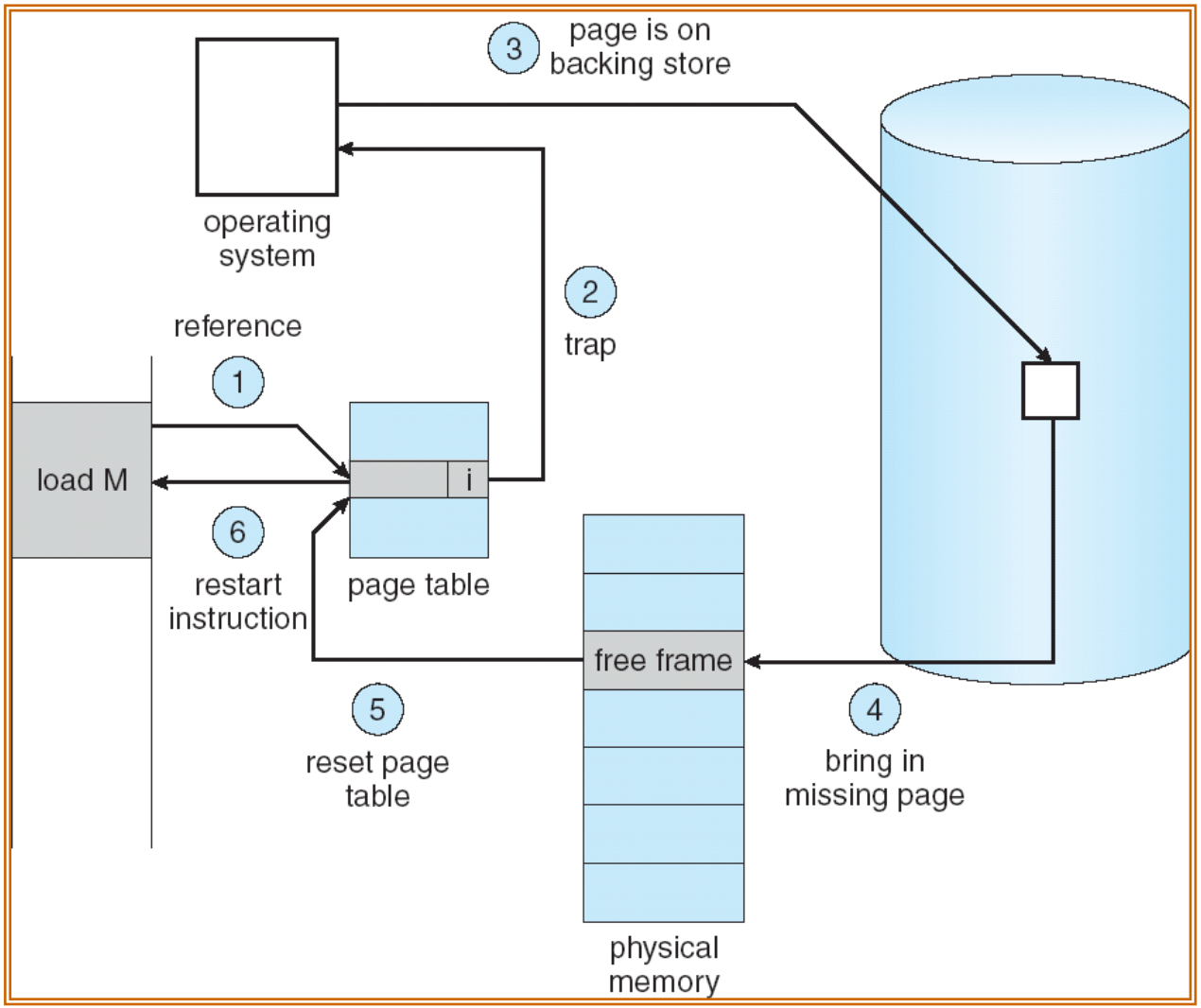
Execution of a Process

- **The OS loads the main memory with some parts of the program (including the starting point)**
- **Each entry in the page table (or segments) has a bit present which indicates whether the page or segment is in main memory**
- **Resident set is the portion of the process in main memory**
- **An interrupt is generated when the logical address refers to a part which is not in the resident set**
  - page fault
- What to do when an address reference falls in a page not in physical memory?
  - Bring that page into memory
  - How?
    - Find it on the disc
    - Get an empty frame.
    - Read the page into frame.
    - Set the valid bit to 1
    - Restart the instruction that caused the page fault.

Demand Paging

- **Bring a page into memory only when it is needed**
- **What does it mean that the page is needed?**
  - Memory address in that page was referenced
- **Fine, but what we need to make it work?**
  - For each page we need to know whether it is in memory or swapped on disc
    - If not, we need to be able to automatically and transparently bring it there
    - If it is on disc, where on disc?

Steps in Handling a Page Fault



Sequence of Events for a Page Fault

1. **Reference is made to an address of a page not loaded into main memory.**
2. **Trap to the OS – an interruption**
   - If the address is legal (page requested is part of the process image).
     - Save registers and state of process in the PCB
     - State is changed to « waiting »
     - PCB placed in the I/O queue.
   - If the address is not legal – terminate the process, i.e. fatal error.
3. **Find the position of the page on the disk.**
4. **Read the page from the disk into a free memory frame (we suppose that one exists)**
   - Execute the disk operations required to read the page.

Sequence of Events for a Page Fault (cont.)

4. **The disk controller completes the transfer and interrupts the CPU.**
   - Note that the process is placed in the Wait state during the I/O operation
5. **The OS updates the contents of the page table for the process that caused the page fault**
   - The process becomes ready.
6. **A some point, the process will execute**
   - The desire page being in memory, the process re-executes the instruction that caused the page fault.

Average memory access time

Suppose that:

- memory access: 100 nanosecs
- paging fault processing time: 25 millisecs = 25,000,000 nanosecs
- p: probability of not finding a page in memory (default)

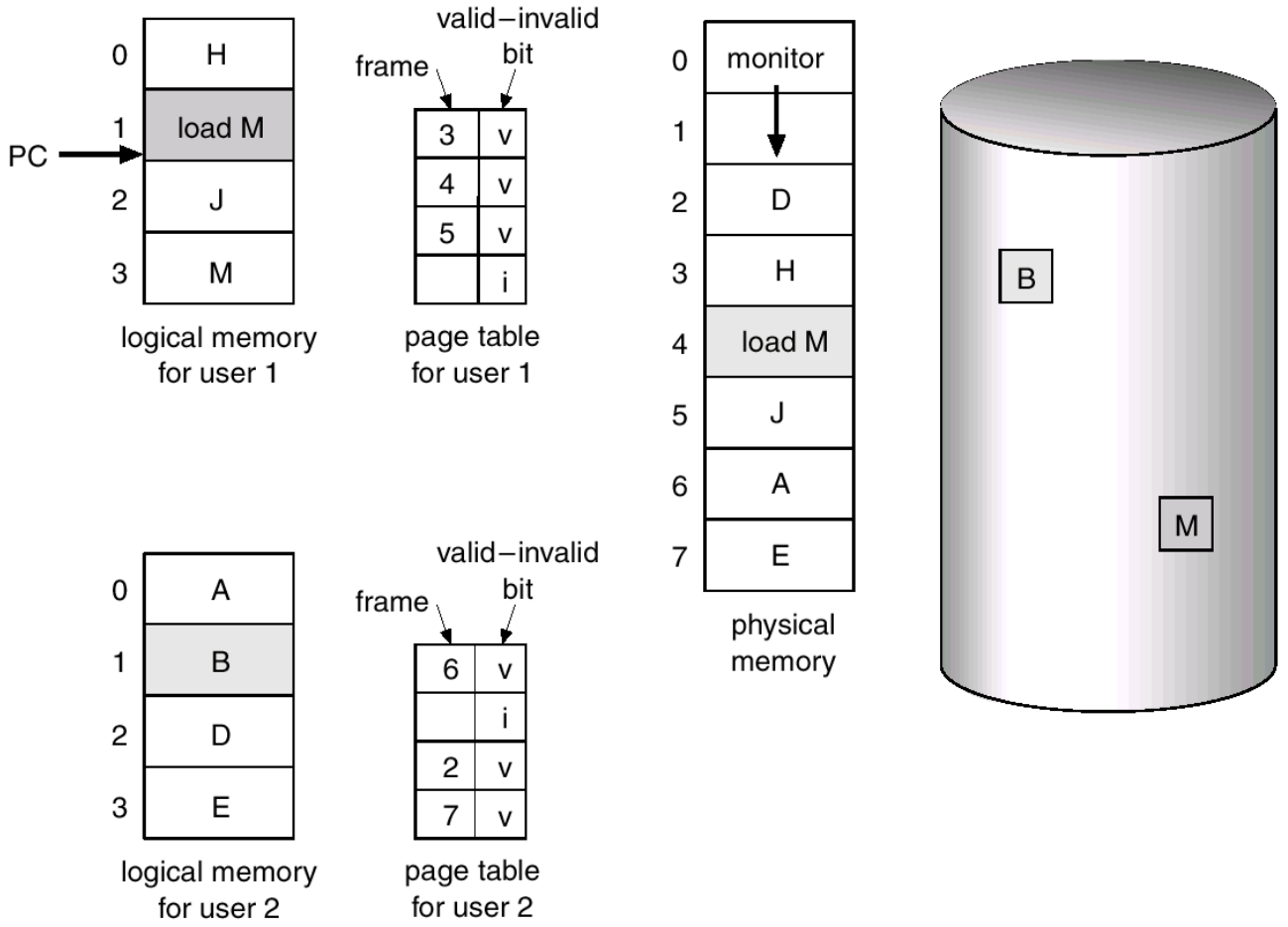
Average memory access time:

(1-p) x 100 + p x 25,000,000 (no fault + fault)

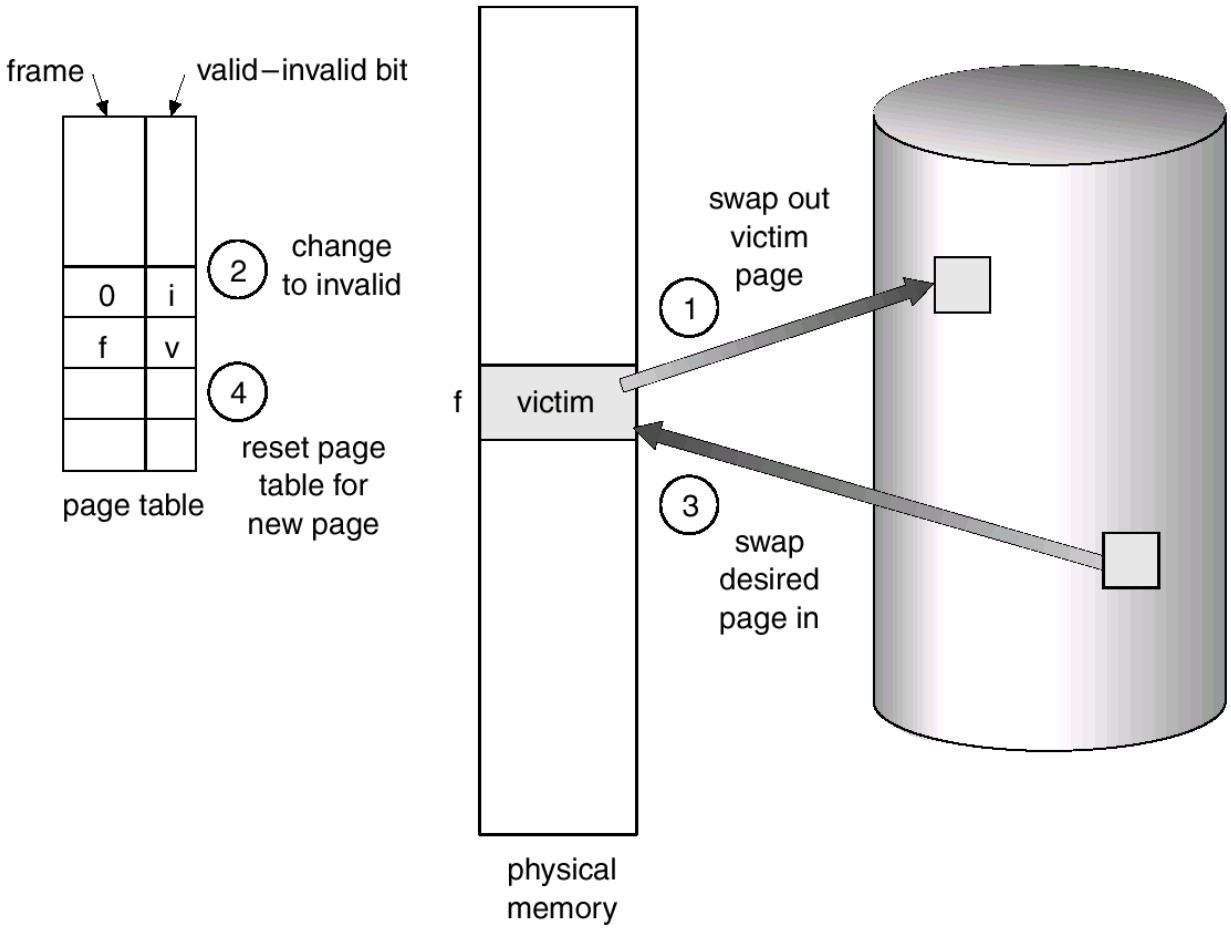
Using the same formula we can determine how many faults we can tolerate, if a certain level of performance is desired (see manual).

Eg. with these params, if the slowdown due to paging cannot exceed 10%, only 1 paging fault can be tolerated for every 2,500,000 memory accesses.

When the RAM is full but we need a page not in RAM



The victim page ...



Basic Page Replacement

- **What if a process requests a new page and there are no free frames in RAM?**
- **It will be necessary to choose a page already in main memory, belonging to the same or to another process, which it is possible to remove from the main memory**
  - the victim!
- **A memory frame will therefore be made available**
- **Obviously, several memory frames cannot be 'victimized':**
  - eg frames containing OS kernel, I/O buffers ...

Dirty bit

- **Does the “victim” really need to be written into secondary memory?**
- **Only if it has not changed since it was brought into main memory.**
  - Otherwise its copy on the disk is still correct.
- **A dirty bit in the page table entry can indicate if the page has changed.**
- **Thus to compute the cost in time of a reference in memory should include the probability that the page is “dirty” and needs to be written to the disk.**

Page replacement algorithms

- **Choose the victim so as to minimize the page defect rate**
  - not easy!!!
- **Page that we won't need in the future? impossible to know!**
- **Page not often used?**
- **Page which has been in memory for a long time ??**

- **etc. we will see...**

Criteria for evaluating algorithms

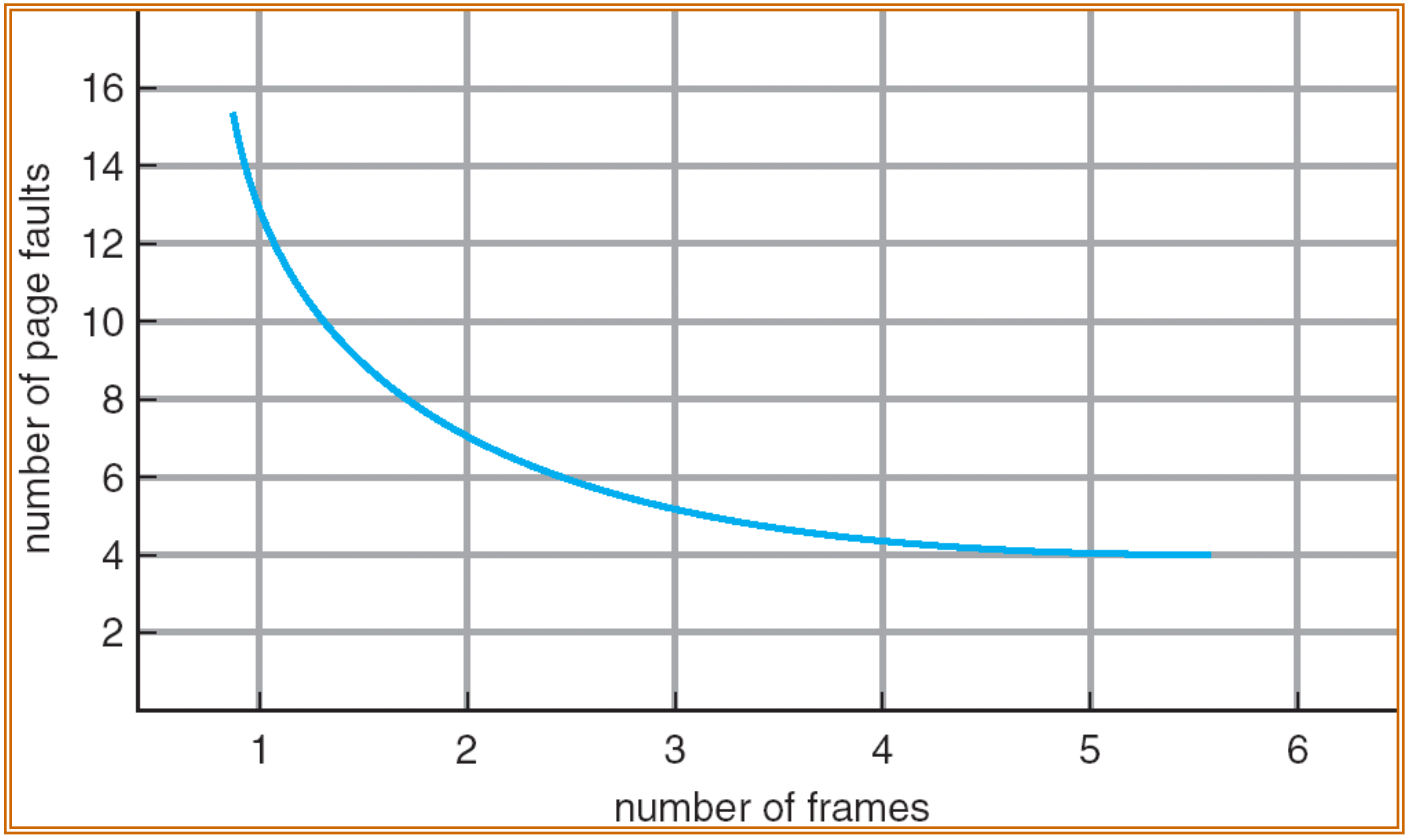
- The algorithms for choosing which pages to replace must be designed so as to minimize page fault rate in the long run
- Should try to keep system overhead to a minimum, e.g. updating tables in memory for each memory access.
- Try to keep expensive hardware to a minimum.

Page Replacement Algorithms

- **Which pages to consider for replacement?**
  - Can we replace kernel pages?
- **Not all pages in main memory can be selected for replacement**
- **Some frames are locked (cannot be paged out):**
  - much of the kernel is held on locked frames as well as key control structures and I/O buffers
- **OK, which unlocked pages to consider for replacement?**
  - Only those of the process that has suffered the page fault
  - Consider all unlocked pages
  - related to the resident set management strategy:
    - how many page frames are to be allocated to each process? We will discuss this later

Graph of Page Faults Versus The Number of Frames

More frames we allocate to the process, less page faults will occur



Explanation and evaluation of algorithms

- **We will explain and evaluate the algorithms using the following page reference strings**

  2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

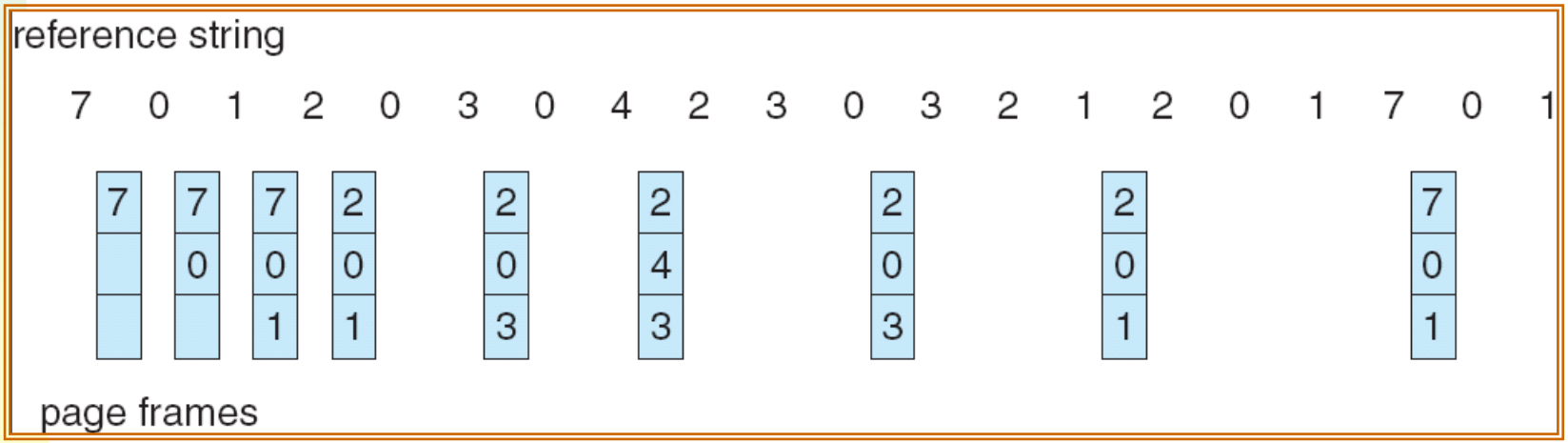
  7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

- **Careful: such sequences are not random.**
  - Remember the locality of reference

- **The evaluation will be made on the basis of this example, obviously not sufficient to draw general conclusions.**

OPT Algorithm

- **Good news:**
  - There is an optimal algorithm that produces the fewest number of page faults
  - **The optimal algorithm (OPT) selects for replacement the the page that will be referenced the latest (i.e.** the page for which the time to the next reference is the longest)

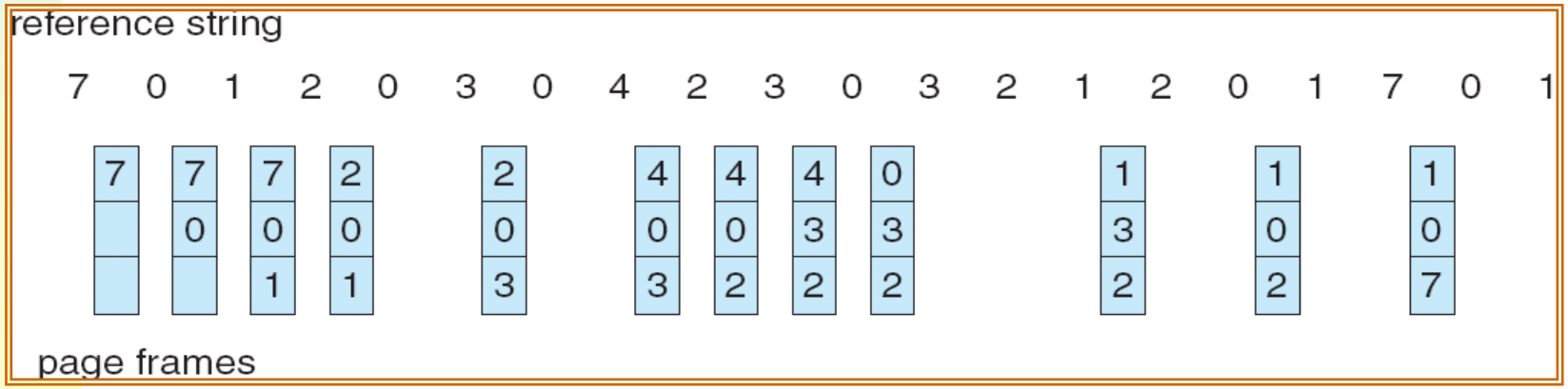


OPT Algorithm

- **Bad news:**
  - impossible to implement (need to know the future)
  - Still useful: serves as a standard to compare with the other algorithms we shall study:
    - Least recently used (LRU)
    - First-in, first-out (FIFO)
    - Second Chance or Clock
    - Counting algorithms

The LRU Algorithm

- Least Recently Used
- Chronological order of use
- Replaces the page that has not been referenced for the longest time
  - By the principle of locality, this should be the page least likely to be referenced in the near future
  - performs nearly as well as the optimal policy

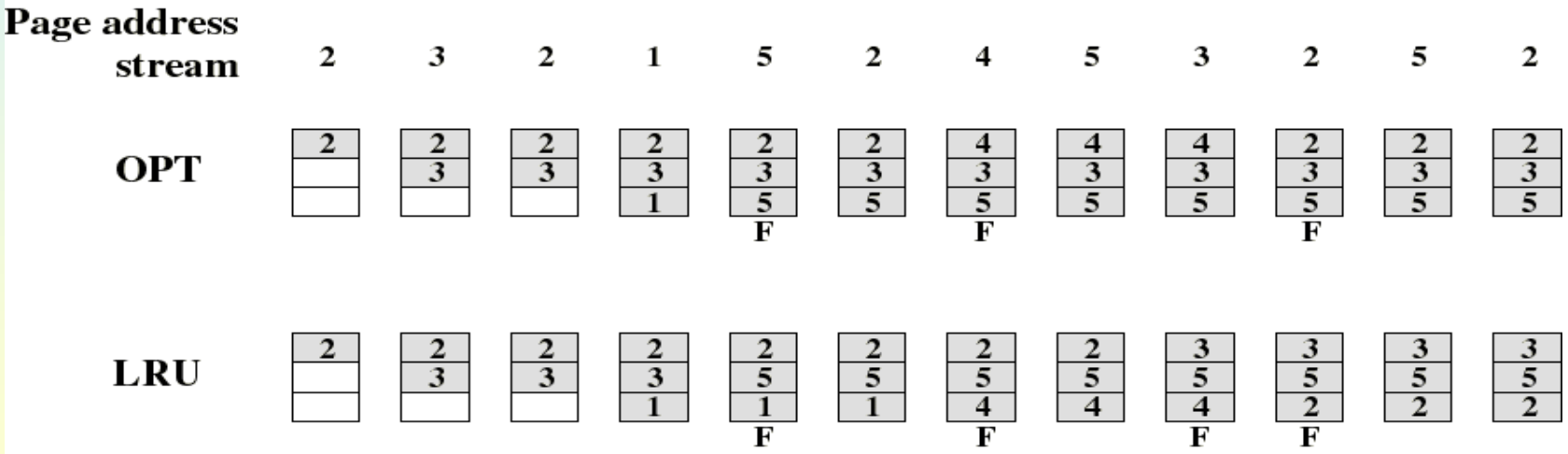


OPT and LRU example

LRU performs nearly as well as OPT:

A 5 page process and there are only 3 physical pages available.

In this example, OPT causes 3 + 3 faults, LRU 3 + 4.



Note on counting page faults

- **When the main memory is empty, each new page we bring in is a result of a page fault**
- **For the purpose of comparing the different algorithms, we are not counting these initial page faults**
  - because the number of these is the same for all algorithms
- **But, in contrast to what is shown in the figures, these initial references are really producing page faults**

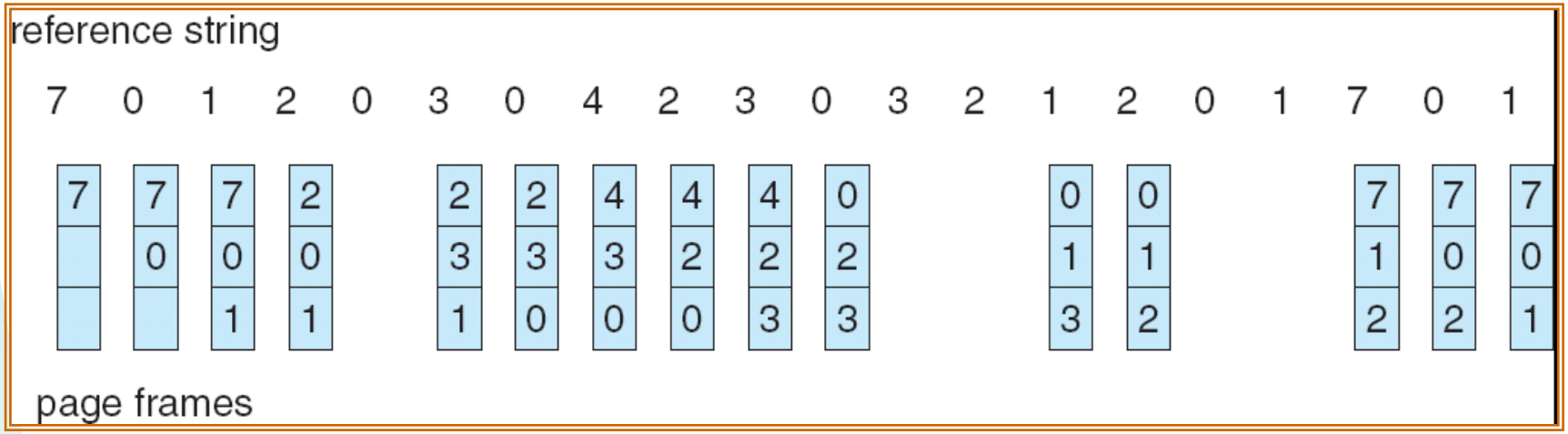
Implementation of the LRU Algorithm

- **What do we need to implement LRU algorithm?**
  - Each page should be tagged (in the page table entry) with the time at each memory reference.
    - This tag should be updated on every reference
    - The LRU page is the one with the smallest time value
      - needs to be searched at each page fault
  - Can be implemented using a stack (linked list)
- **Both approaches would require expensive hardware and a great deal of overhead.**
  - Consequently very few computer systems provide sufficient hardware support for true LRU replacement algorithm
  - Other algorithms are used instead such as LRU *approximations*

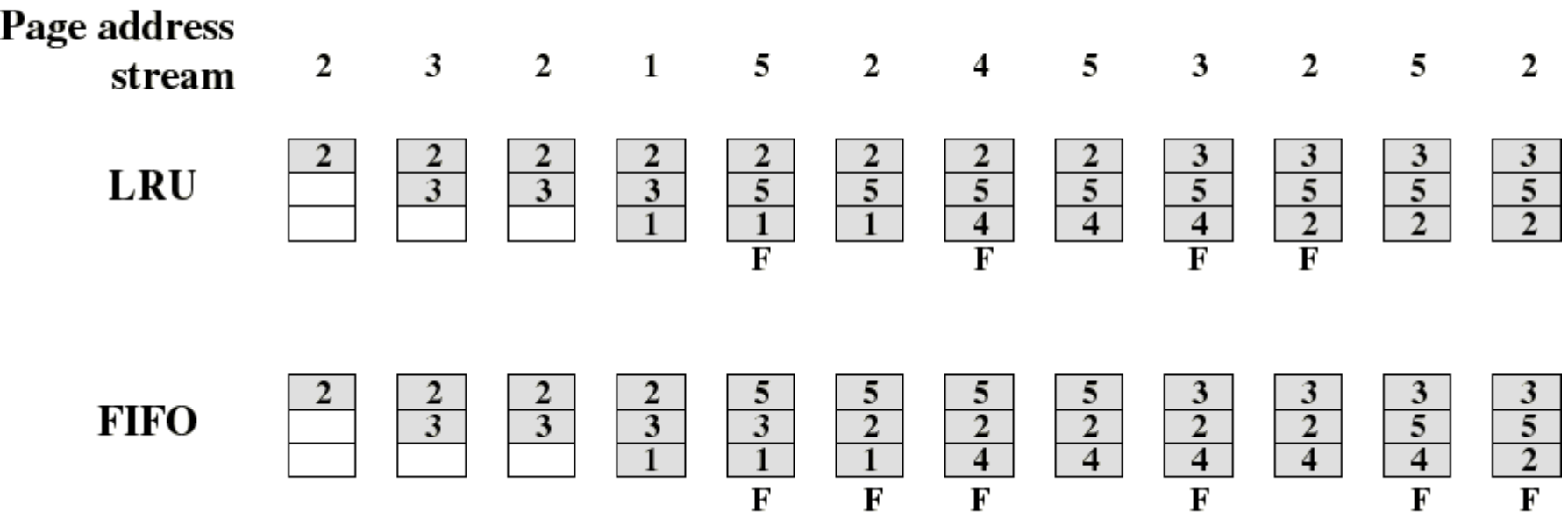
The FIFO Algorithm

- **OPT is impossible, LRU is too costly, let's try something simple!**
- **Idea: a page which has been in memory for a long time has had its chance to execute**
- **Treats page frames allocated to a process as a circular buffer:**
  - When the buffer is full, the oldest page is replaced. Hence: first-in, first-out
    - This is not necessarily the same as the LRU page
    - "The oldest page" means the page in memory the longest (for LRU, in terms of page reference).
  - Simple to implement
    - requires only a pointer that circles through the frames of the process
- **BUT: A frequently used page is often the oldest, so it will be repeatedly paged out by FIFO**

FIFO Example



Comparison of FIFO with LRU (Stallings)



- **Unlike FIFO, LRU recognizes that pages 2 and 5 are used frequently**
- **FIFO's performance is less good:**
  - in this case, LRU = 3 + 4, FIFO = 3 + 6

Conceptual problem with FIFO

- **The first pages brought into memory are often useful throughout the execution of a process!**
  - global variables, main program, etc.
- **Which shows a problem with our way of comparing methods based on a random sequence:**
  - references to pages in a real program will not be really random

Belady's anomaly

- **For some algorithms, in some cases it could have more faults with more memory!**
  - E.g., FIFO, but not LRU, OPT,CLOCK
  - (Ex. referral string 1,2,3,4,1,2,5,1,2,3,4.5)

Situation considered normal

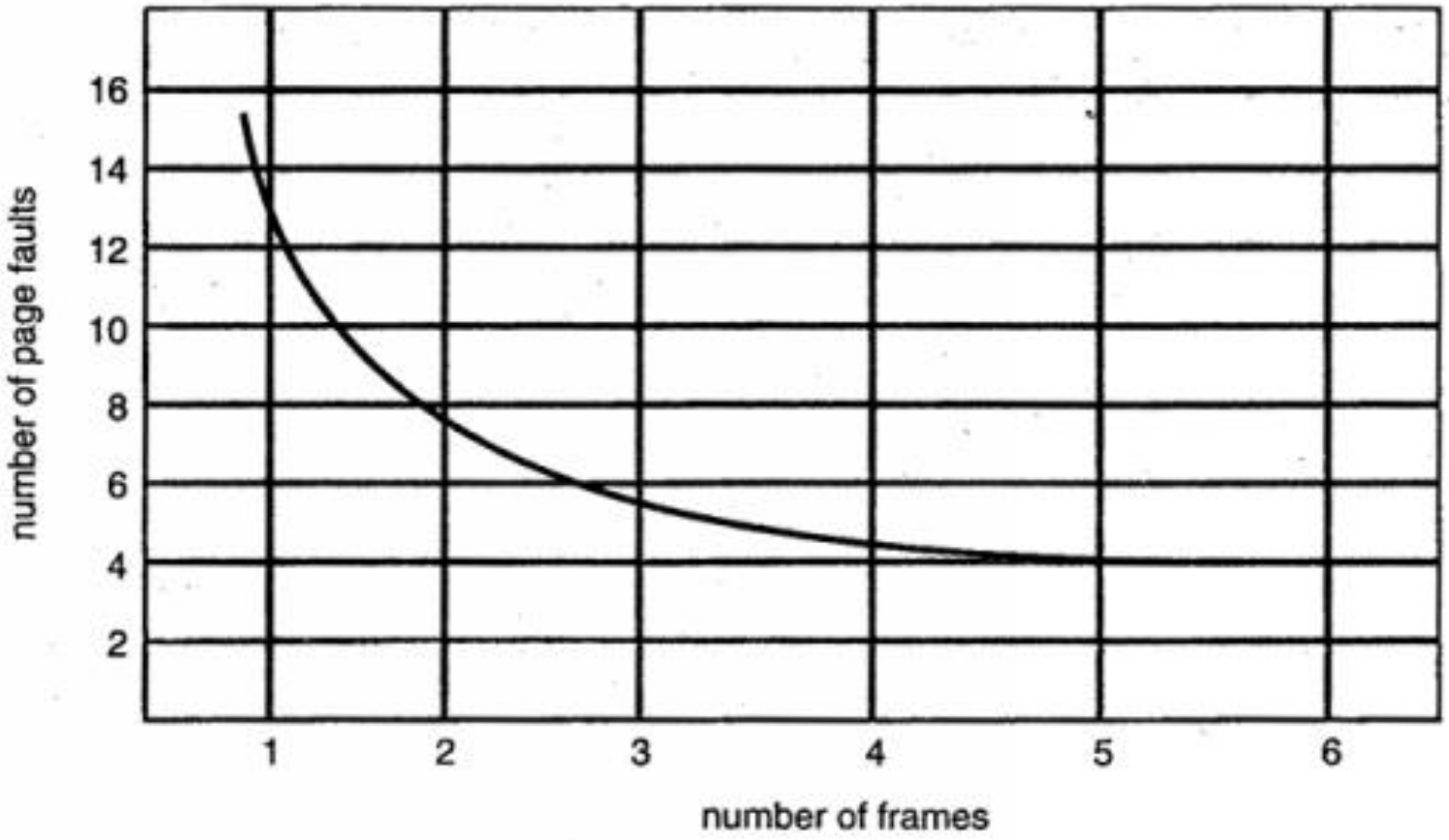
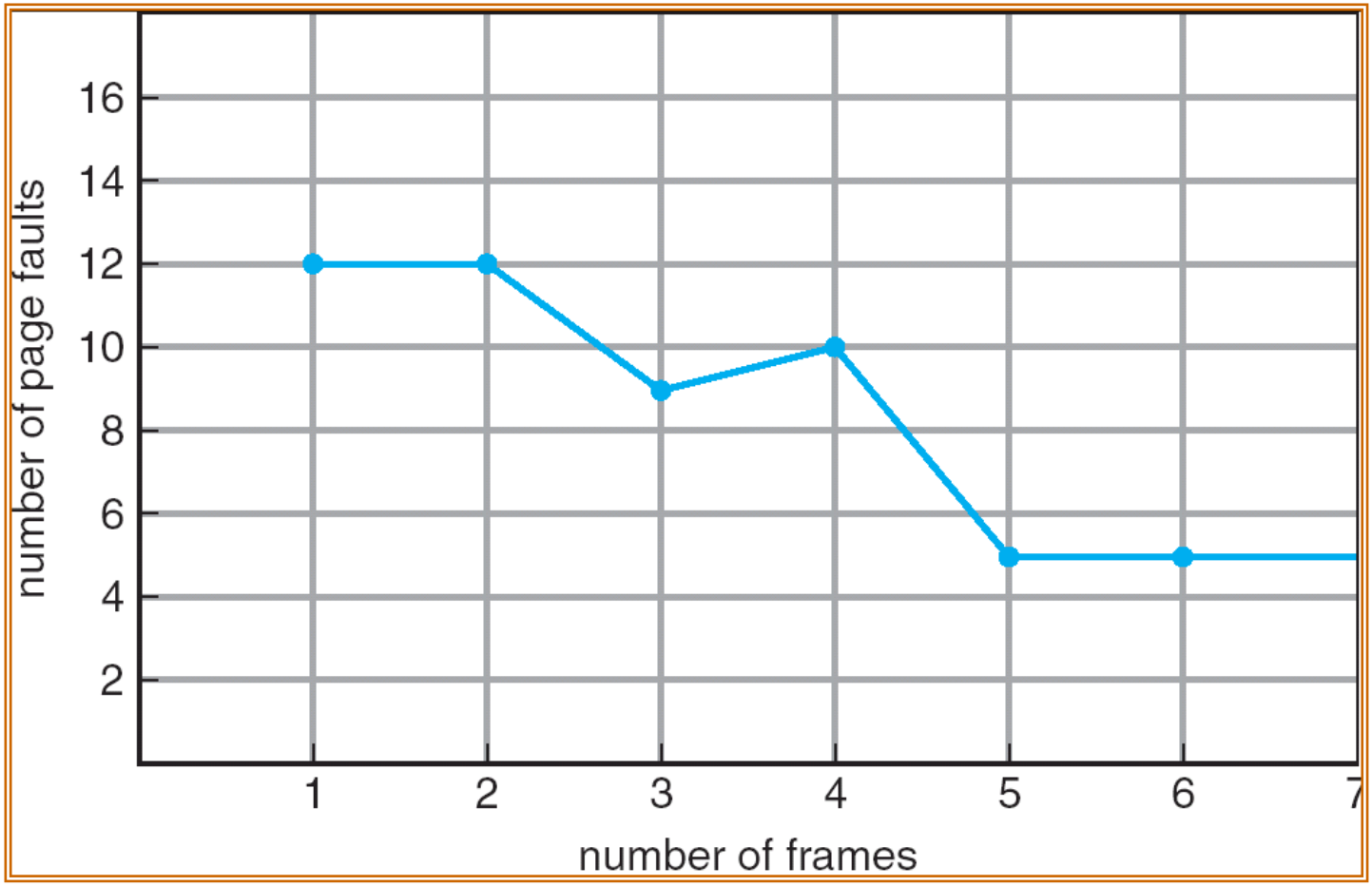


Figure 10.7 Graph of page faults versus the number of frames.

Belady's Anomaly

Number of page faults does not necessarily decrease with increased number of frames:



Belady's Anomaly in FIFO Algorithm

- **Chain of References: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
- **3 frames (3 pages can be in memory at a time per process)**

1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- **4 frames**

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

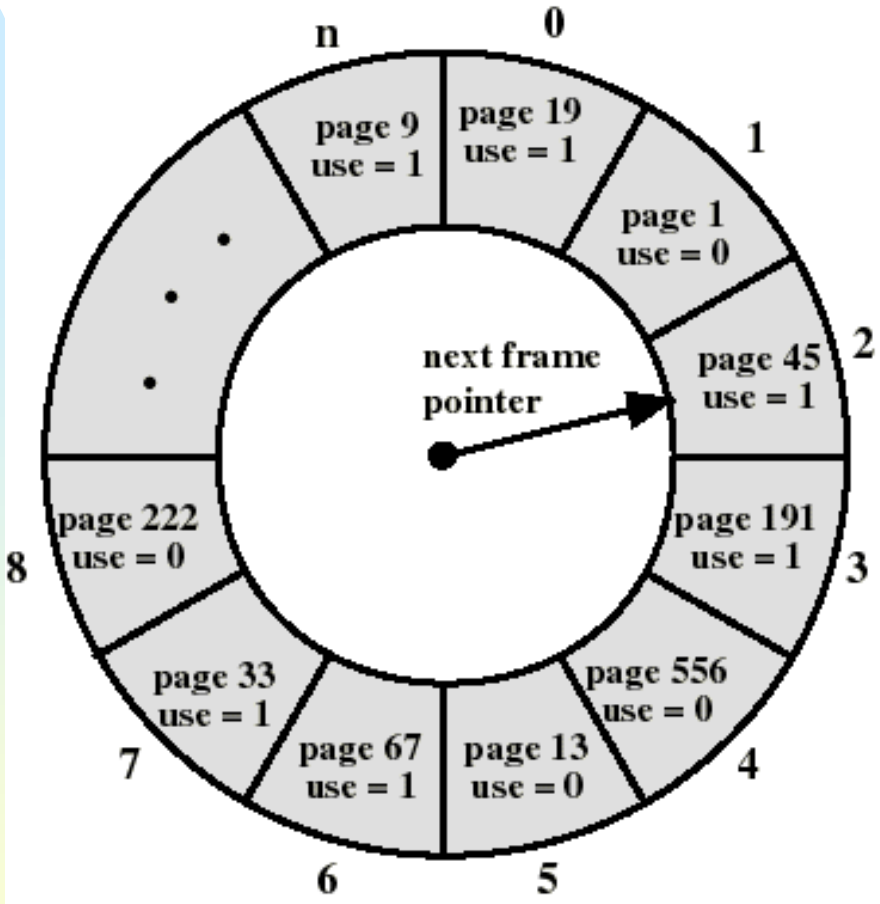
Page replacement algorithms

- **OPT**
  - Impossible to implement
- **Let's approximate it with LRU**
  - But LRU is still too costly
- **Let's try something simple – FIFO**
  - Does not work well enough
- **Any further ideas?**
  - Try to approximate the approximation
  - Approximate LRU
    - Clock algorithm
    - Counting algorithms

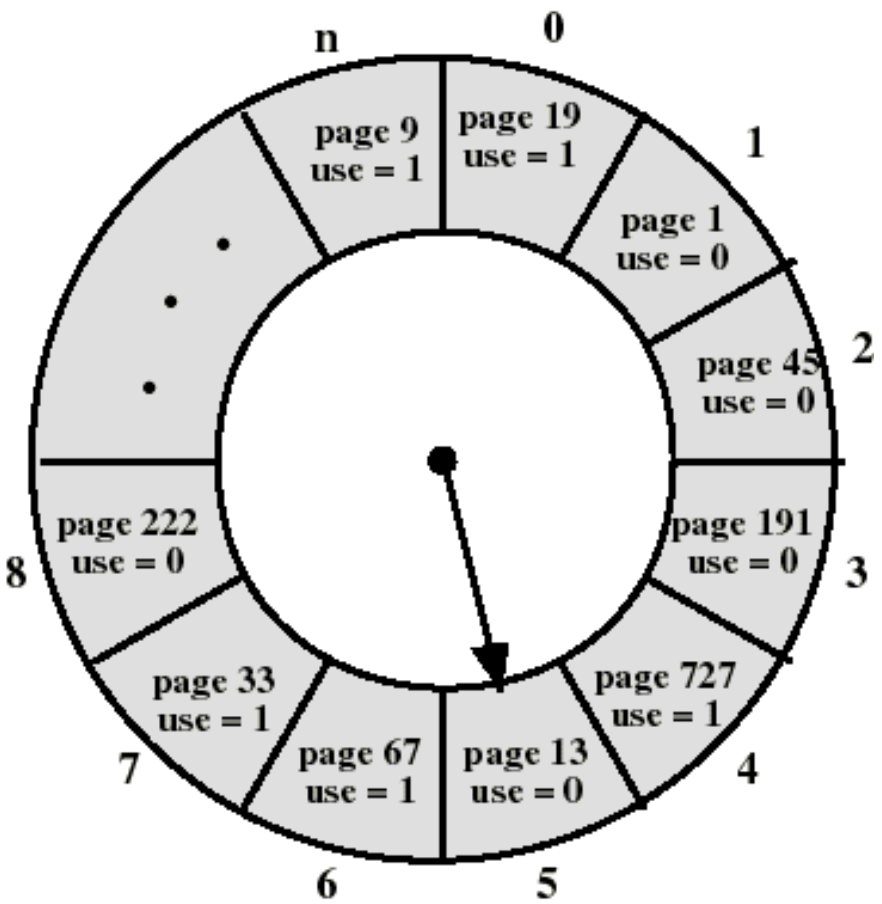
The Clock Algorithm (second chance algorithm)

- **The set of frames candidate for replacement is considered as a circular buffer (similar to FIFO)**
  - When a page is replaced, a pointer is set to point to the next frame in buffer
- **A use bit for each frame is set to 1 whenever**
  - a page is first loaded into the frame
  - the corresponding page is referenced
- **When it is time to replace a page, the first frame encountered with the use bit set to 0 is replaced.**
  - During the search for replacement, each use bit set to 1 is changed to 0
- **Gives each page a second chance to prove that it is useful**
  - By not replacing it when considered for the first time
  - By not replacing if it was referenced after it has been considered at the last page fault

Clock algorithm: an example (Stallings).



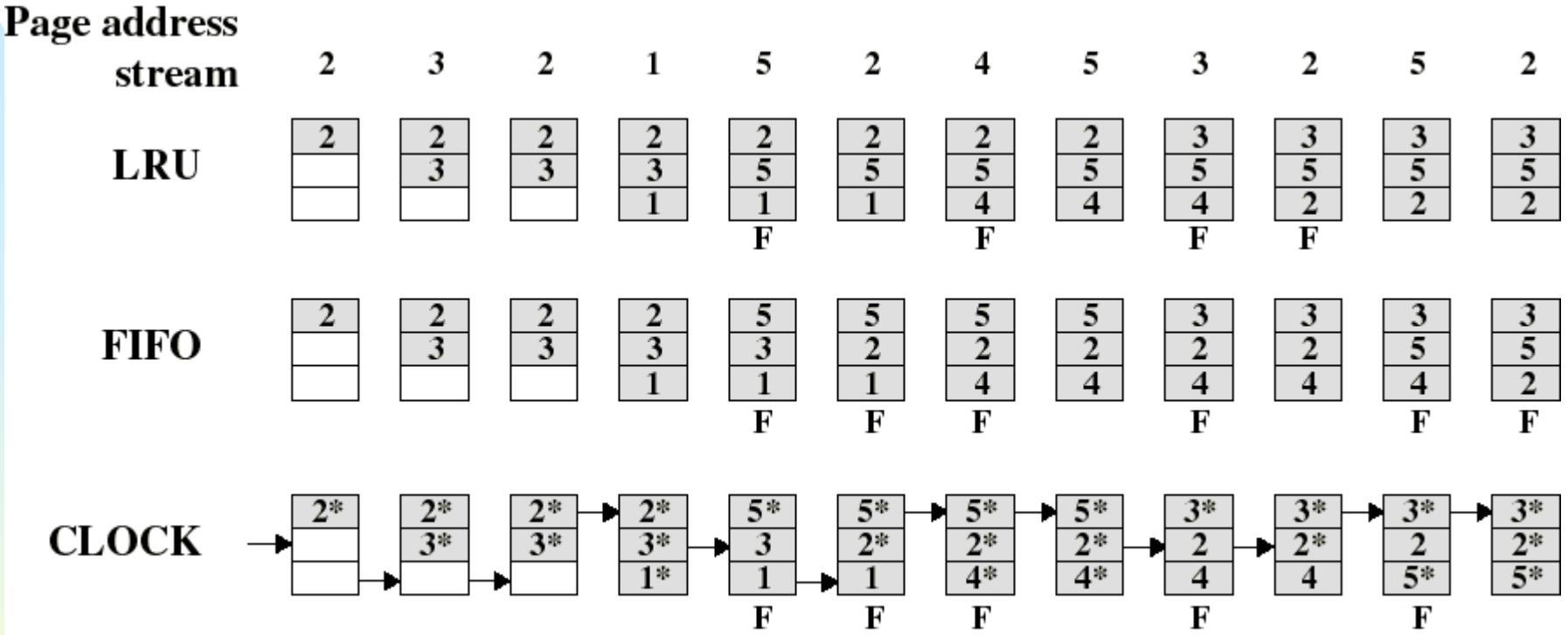
(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement

Page 727 is loaded in frame 4.
The next. victim is 5, then 8.

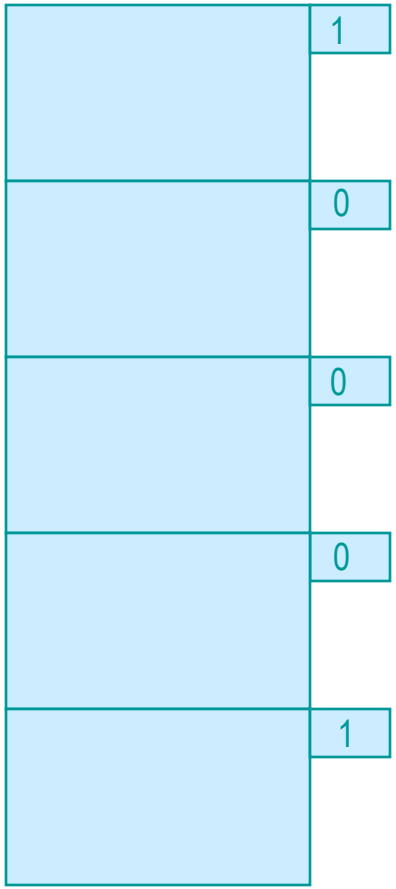
Comparison of Clock with FIFO and LRU



- Asterisk indicates that the corresponding use bit is set to 1
- Clock protects frequently referenced pages by setting the use bit to 1 at each reference
- LRU = 3+4, FIFO = 3+6, Clock = 3+5

Additional Hardware for the CLOCK Algorithm

- **Each bloc of memory has a use bit**
- **When the contents of the bloc has been used, the bit is set to 1 by the hardware.**
- **The OS can examine this bit**
  - If 0, the page can be replaced
  - If 1, it sets it to 0.



Comparaison: Clock, FIFO et LRU

- **Simulations show that clock has almost the same performance as LRU**
  - **Variations of the clock are implemented in real systems**
- **When candidate pages for replacement are local to the process that invoked the page fault and the number of frames allocated are fixed, experiments show:**
  - **If few (6 to 8) frames are allocates, the number of page faults produced by FIFO is almost double that produced by LRU, and that produced by CLOCK is somewhere in between.**
  - **The difference between LRU and CLOCK tends to disappear as (over 12) frames are allocated.**

Counting Algorithms

- **Keep a counter of the number of references that have been made to each page**
- LFU Algorithm: **replaces page with smallest count**
- MFU Algorithm: **based on the argument that the page with the smallest count was probably just brought in and has yet to be used**
- **Implementation of these algorithms are expensive and rarely used.**

Using a stack

- **When a page is used, it is put on top of the stack.**
  - so the most recently used page is always at the top,
  - the least recently used is still at the bottom
- **Good implementation of the principle of locality, however ...**
- **The stack must be implemented by hardware, as we cannot tolerate the execution of a program every time a page is used**
- **So not practical**

Page Buffering

- **Pages to be replaced are kept in main memory for a while to guard against poorly performing replacement algorithms such as FIFO**
- **Two lists of pointers are maintained: each entry points to a frame selected for replacement**
  - a free page list for frames that have not been modified since brought in (no need to swap out)
  - a modified page list for frames that have been modified (need to write them out)
- **A frame to be replaced has a pointer added to the tail of one of the lists and the present bit is cleared in corresponding page table entry**
  - but the page remains in the same memory frame

Page Buffering

- **At each page fault the two lists are first examined to see if the needed page is still in main memory**
  - If it is, we just need to set the present bit in the corresponding page table entry (and remove the matching entry in the relevant page list)
  - If it is not, then the needed page is brought in, it is placed in the frame pointed by the head of the free frame list (overwriting the page that was there)
    - the head of the free frame list is moved to the next entry
  - (the frame number in the page table entry could be used to scan the two lists, or each list entry could contain the process id and page number of the occupied frame)
- **The modified list also serves to write out modified pages in batch (rather than individually)**

Frame Allocation

- To execute, a process needs a minimal number of memory frames.
  - For example, a few instructions may require a number of pages for execution (code, data, stack).
- It is easy to see that when a process receives few frames, it will generate an excessive number of page faults and be slowed considerably.
- How to ensure that a process is allocate its minimum
  - Equal allocation: each process has an equal portion of physical memory.
  - Proportional allocation: each process is allocate according to its size
  - The criteria should be more related to the pages needed: see working set.

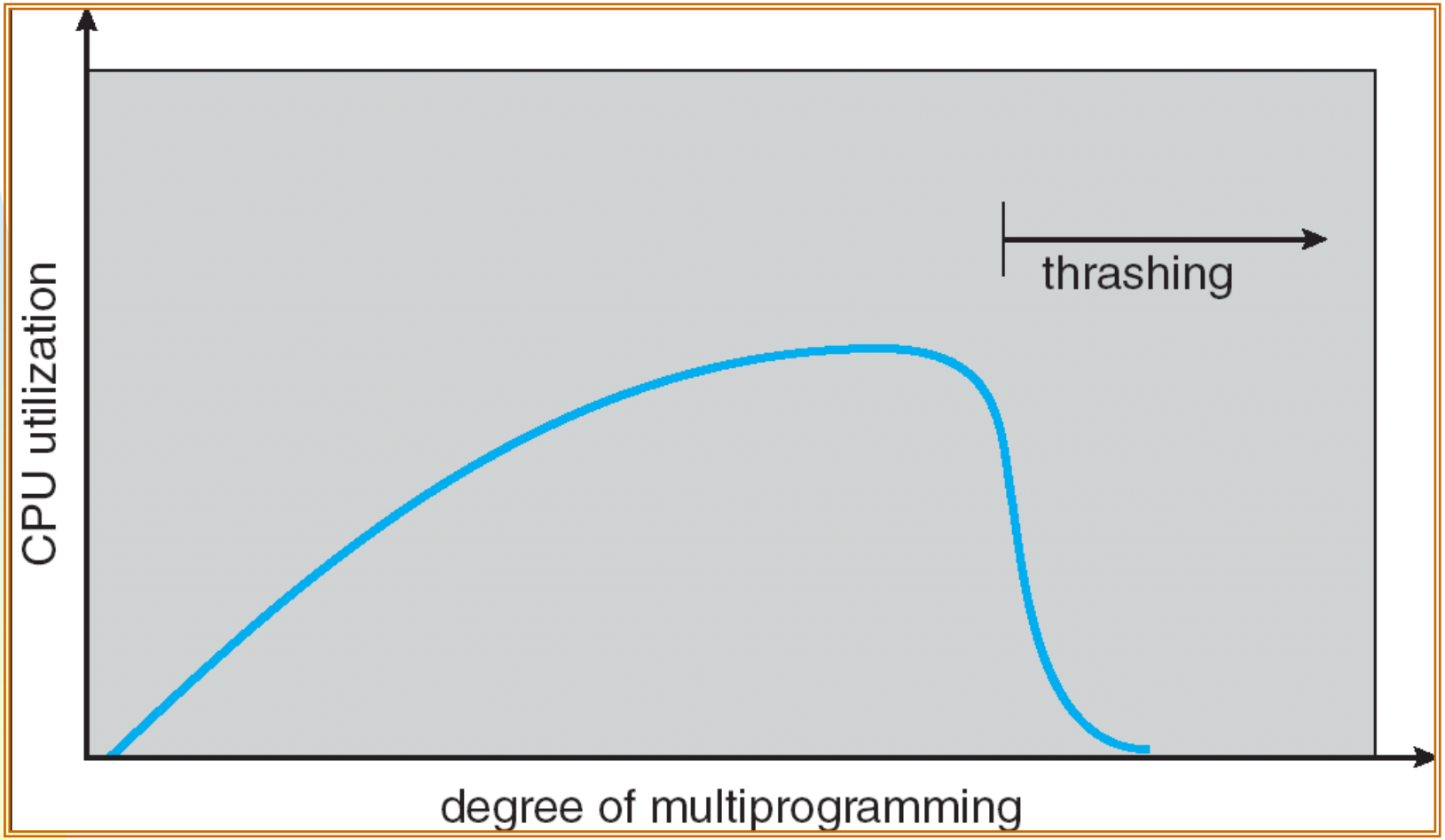
Global or local allocation

- global: the `victim` is taken from any process
- local: the `victim` is taken from the process that needs the page

Thrashing

- **If not enough memory is allocated to a process so that it generates many page faults, the process spends it time in the I/O queues.**
- **If this situation is generalized to many processes, the CPU becomes under utilized.**
- **The OS can help remedy the situation by increasing the level of multiprogramming**
  - More processes in memory
  - Less memory per process
  - More page faults
- **Disaster: thrashing**
  - The system is so busy doing I/O of pages, it not longer can complete any useful work.

Thrashing



The reason for the thrashing

- **Each process needs a certain number of pages to run efficiently.**
- **The number of pages that the set of processes needs now exceeds the number of available memory frames**
- **We really need to avoid thrashing!**
- **How to do that?**
  - By managing the resident set sizes of processes
  - By controlling the degree of multiprogramming
    - Load control

Resident Set Size

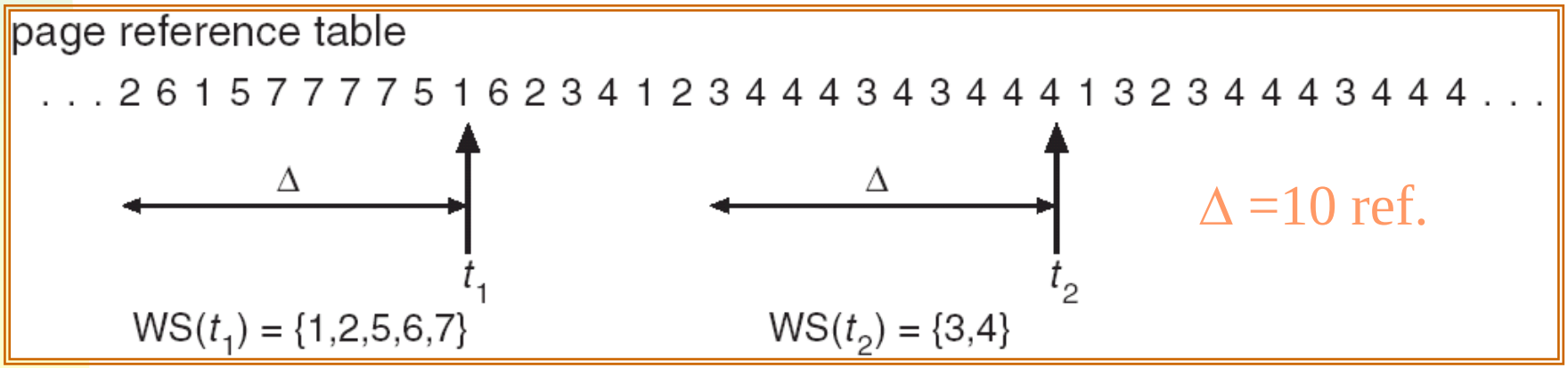
- **The OS must decide how many page frames to allocate to a process**
  - large page fault rate if too few frames are allocated
  - low multiprogramming level if to many frames are allocated

Working Set

- The working set of a process at a point in execution consists of the set of pages the process requires for execution without too many page faults.

The Working Set Strategy

- **Is a variable-allocation method with local scope based on the assumption of locality of references**
- **The working set for a process at time t, WS(t), is the set of pages that have been referenced in the last Δ virtual time units**
  - virtual time = time elapsed while the process was in execution (in terms of memory references)
  - Δ is a window of time
  - WS(t) is an approximation of the program's locality



Δ = 4 would give the same results for t_2!

The Working Set Strategy

- **The working set of a process first grows when it starts executing**
- **then stabilizes by the principle of locality**
- **it grows again when the process enters a new locality (transition period)**
  - up to a point where the working set contains pages from two localities
- **then decreases after a sufficient long time spent in the new locality**

Working Set and Page Faults

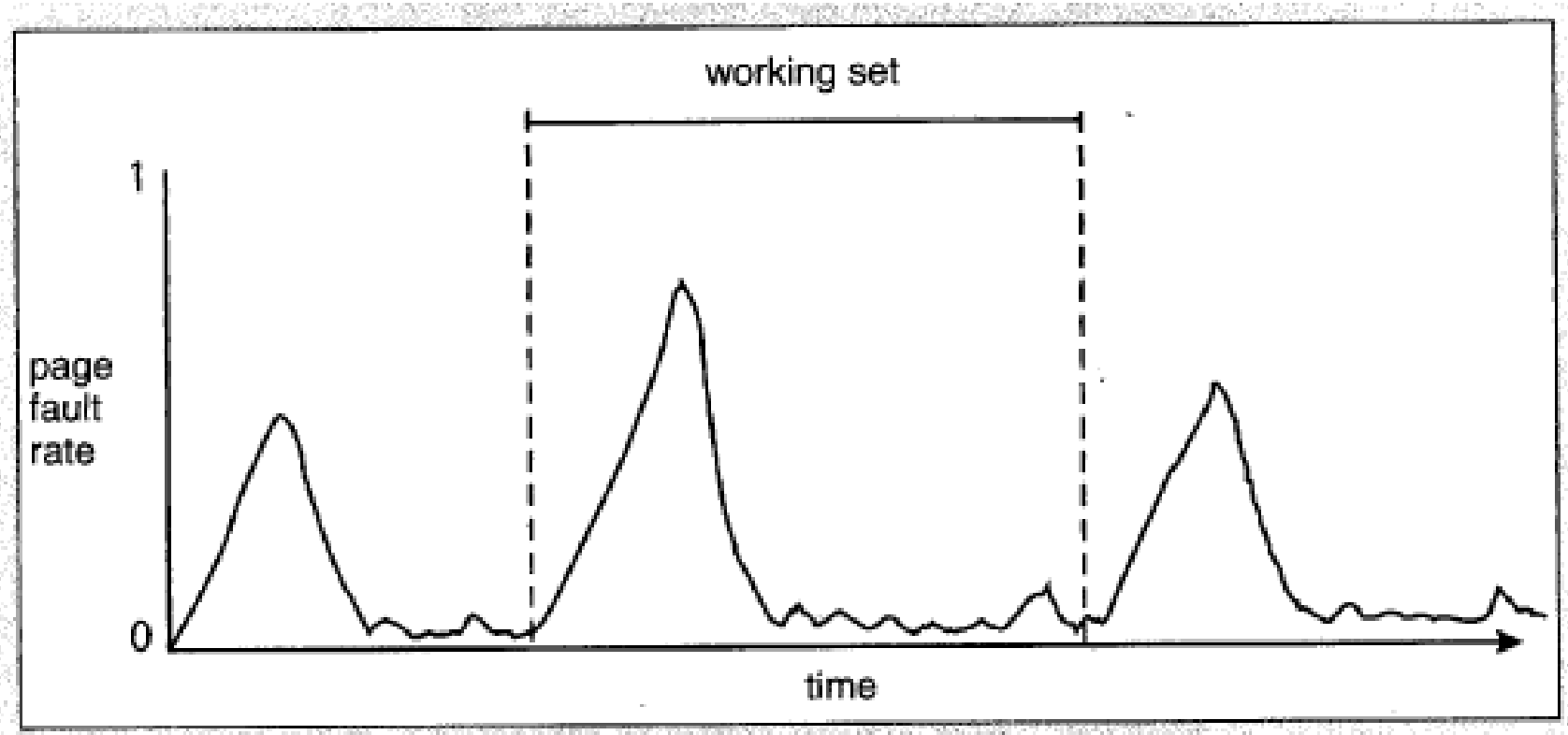


Figure 9.22 Page fault rate over time.

The Working Set Strategy

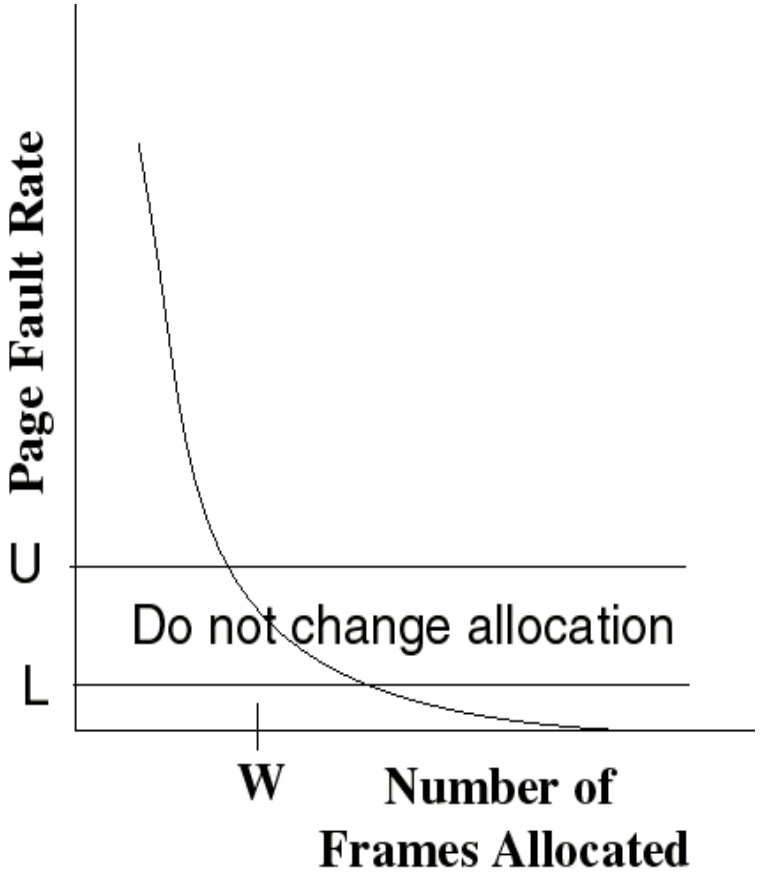
- **the working set concept suggest the following strategy to determine the resident set size**
  - Monitor the working set for each process
  - Periodically remove from the resident set of a process those pages that are not in the working set (apply LRU).
  - When the resident set of a process is smaller than its working set, allocate more frames to it
    - If not enough free frames are available, suspend the process (until more frames are available); ie: a process may execute only if its working set can be loaded in main memory

The Working Set Strategy

- **Practical problems with this working set strategy:**
  - measurement of the working set for each process is impractical
    - necessary to stamp the referenced page with virtual time at every memory reference
    - necessary to maintain a time-ordered queue of referenced pages for each process
  - the optimal value for Δ is unknown and time varying
- Any solution?**:**
  - rather than monitoring the working set, monitor the page fault rate!

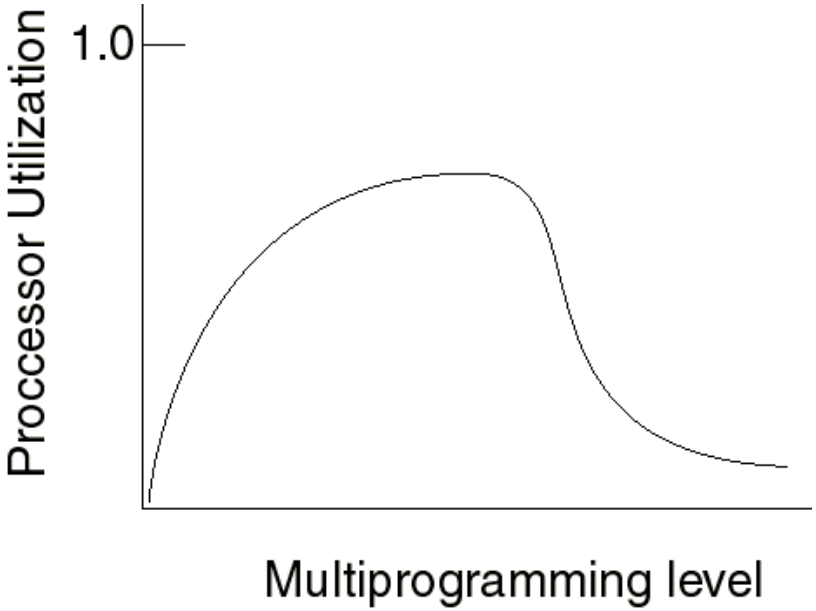
The Page-Fault Frequency Strategy

- Define an upper bound U and lower bound L for page fault rates
- Allocate more frames to a process if fault rate is higher than U
- Allocate less frames if fault rate is < L
- The resident set size should be close to the working set size W
- We suspend the process if the PFR > U and no more free frames are available



Load Control

- **Determines the number of processes that will be resident in main memory (ie: the multiprogramming level)**
  - Too few processes: often all processes will be blocked and the processor will be idle
  - Too many processes: the resident size of each process will be too small and flurries of page faults will result: thrashing



Load Control

- **A working set or page fault frequency algorithm implicitly incorporates load control**
  - only those processes whose resident set is sufficiently large are allowed to execute
- **Another approach is to adjust explicitly the multiprogramming level so that the mean time between page faults equals the time to process a page fault**
  - performance studies indicate that this is the point where processor usage is at maximum

Process Suspension

- **Explicit load control requires that we sometimes swap out (suspend) processes**
- **Possible victim selection criteria:**
  - Faulting process
    - this process may not have its working set in main memory so it will be blocked anyway
  - Last process activated
    - this process is least likely to have its working set resident
  - Process with smallest resident set
    - this process requires the least future effort to reload
  - Largest process
    - will yield the most free frames

Locking Pages in Memory

- **Some pages must be locked in memory, e.g. those containing the OS kernel**
- **It is also essential to lock in memory pages on which there is execution of I/O**
- **This can be achieved with a `lock` bit on the memory frame.**
  - this bit means that this frame cannot be selected as `victim`

Real time systems

- **With virtual memory, the execution times of a process becomes less predictable**
  - unexpected delays due to pagination
- **So hard real time systems rarely use virtual memory**

Combination of techniques

- **The real OS use the techniques that we studied in *combination*, eg**
  - Linux uses paging (the smallest amount of allocable memory is a page)
  - other systems use fixed partitions with paging, which can be done in several ways:
    - Divide *real* memory in fixed partitions, assign each partition to one or more processes, then paginate a process in the partition assigned to it
    - Divide *virtual* memory in partitions, assign each partition to one or more process, then use the appropriate technique for each process in its partition
- **Real OS are complex and varied, but the principles explored in this course form the basis.**

Conclusions 1

- **It is highly desirable that the user address space can be much larger than the RAM memory address space.**
- **The programmer will therefore be freed from the concern of managing his memory allocation.**
  - however, he should seek to maximize the locality of his process
- **Virtual memory also allows more processes to be running.**
  - CPU, busier I/O

Conclusions 2

- **The problem of deciding the victim page is not easy.**
  - The best algorithms are impossible or difficult to implement
  - However in practice the FIFO algorithm is acceptable.

Conclusions 3

- **Make sure that each process has enough pages in physical memory to run efficiently**
  - risk of trashing
- **The working set model expresses the requirements well, however it is difficult to implement**
- **More pragmatic solution, where one decides to give + or - of memory to the processes according to their rate of pagination faults**
- **In order for these memory management mechanisms to be efficient, several types of mechanisms are desirable in the hardware.**


Thank You!
Ευχαριστώ
ขอบคุณ
Vielen Dank
Merci
Teşekkürler
DMnvwd
شكراً
متشكرم
Gracias
Grazie
Bedankt
Dankie
THANK YOU
Köszönettel
Obrigado!
شکریا
Díky
謝謝
GADDA GUEY
WAD MAHAD
SAN TAHAY
감사합니다
Urakoze